A Rabin-Williams Signature Scheme

Adam Langley (agl@google.com)
(Version 20080924)

1. Introduction

This is example code for a Rabin-Williams public-key signature scheme designed to provide high speed verification and small signatures. Key points:

1. Fast verification: about $7\mu s$ for a short message on a 2.33GHz Core2 (1024 bit key). RSA 1024 on the same hardware is about 4x slower.
2. Small(er) signatures: signatures are half the size of RSA signatures for the same key strength.
3. A hash generic attack is provably equivalent to factoring.

This scheme is parameterised over the length of the public key which will be referred to as s in this text. The code itself assumes $s = 1024$, although this is easy to change.

This is simply an exposition of the work of Rabin, Williams, Bernstein, Bleichenbacher and others. Any artifice found here is theirs, any mistakes are mine.

⟨ Preamble 3 ⟩⟨ Key generation 4 ⟩⟨ Signature generation 13 ⟩⟨ Signature Verification 29 ⟩

2. Standard includes

We use a few common C header files. `<stdint.h>` is used to get **uint8_t**, which this author prefers to **unsigned char**.

⟨ Standard includes 2 ⟩ ≡
```
#include <stdint.h>
#include <limits.h>
#include <stdlib.h>
#include <string.h>
```
This code is used in section 3.

3. Preamble

We use the GNU Multiple Precision Arithmetic Library (GMP, http://gmplib.org) for integer functions and OpenSSL for its SHA512 implementation. This code follows the eBACS API for signature schemes (http://bench.cr.yp.to).

Use of GMP may be problematic for some as it doesn't handle out-of-memory conditions gracefully. Readers are directed to the GMP manual for more details.

We assume a function *randombytes* which fills a specified buffer with random data. Users are free to implement this function however they wish. Typically it is done by using /dev/urandom.

```
#define SECRETKEYBYTES (64 + 64 + 128 + 1 + 8)
#define PUBLICKEYBYTES 128
#define BYTES 65
  format mpz_t int
  format uint8_t int
```

⟨ Preamble 3 ⟩ ≡
```
  ⟨ Standard includes 2 ⟩
#include <gmp.h>
#include <openssl/sha.h>
    extern void randombytes(uint8_t *buffer, unsigned long long length);
```
This code is used in section 1.

4. Key Generation.

Let p be a prime $\in 3 + 8\mathbb{Z}$. Let q be a prime $\in 7 + 8\mathbb{Z}$. The public key is $n = pq$. In order to avoid an additional reduction in verification, we also require that $n > 2^{s-8}$.

⟨ Key generation 4 ⟩ ≡
  ⟨ Random prime generation 5 ⟩
  ⟨ Hash function 16 ⟩
  ⟨ HMAC function 27 ⟩
  ⟨ Extended Euclid 12 ⟩
  ⟨ Key pair function 6 ⟩

This code is used in section 1.

5. Random prime generation

We require the ability to generate primes of a given size which are congruent to a given value modulo 8. We don't use GMP's built in random generation, preferring to use the *randombytes* function, declared above. The returned value is prime with high probability. We use a Miller-Rabin probabilistic primality test with 32 iterations thus bounding the probability of returning a composite $\leq 2^{-64}$.

This function generates a random prime, p at most *size* bits long and such that $p \equiv mod8 \bmod 8$. The result is written to n, which should not be initialised upon entry. *size* must be less than 2049.

⟨ Random prime generation 5 ⟩ ≡

```
static void init_random_prime(mpz_t n, unsigned size, unsigned mod8)
{
  uint8_t buffer[256];
  const unsigned bytes = size >> 3;
  if (bytes > sizeof (buffer)) abort();
  mpz_init2(n, bytes);
  for ( ; ; ) {
    randombytes(buffer, bytes);
    mpz_import(n, bytes, 1, 1, 0, 0, buffer);
    mpz_setbit(n, 0);
    if (mod8 & 2) {
      mpz_setbit(n, 1);
    }
    else {
      mpz_clrbit(n, 1);
    }
    if (mod8 & 4) {
      mpz_setbit(n, 2);
    }
    else {
      mpz_clrbit(n, 2);
    }
    if (mpz_probab_prime_p(n, 32)) break;
  }
}
```

This code is used in section 4.

6. Generating a key pair

The *keypair* function generates a keypair and stores the public key in `pk[0]`, `pk[1]`, ..., `pk[PUBLICKEYBYTES - 1]`, stores the secret key in `sk[0]`, `sk[1]`, ..., `sk[SECRETKEYBYTES_ - 1]` and returns 0.

```
⟨ Key pair function 6 ⟩ ≡
  int crypto_sign_rwb0fuz1024_gmp_keypair(uint8_t *pk, uint8_t *sk)
  {
    mpz_t p, q, n;
    ⟨ Pick primes 7 ⟩
    ⟨ Chinese remainder precomputation 8 ⟩
    ⟨ Generate HMAC secret 9 ⟩
    ⟨ Keypair serialisation 10 ⟩
    ⟨ Keypair cleanup 11 ⟩
    return 0;
  }
```

This code is used in section 4.

7. Picking primes

We generate a pair of 512-bit primes, p and q where $p \in 3 + 8\mathbb{Z}$ and $q \in 7 + 8\mathbb{Z}$. We also test that $n = pq > 2^{1016}$ by looking for a true bit in the top 8 bits of n.

```
⟨ Pick primes 7 ⟩ ≡
  for ( ; ; ) {
    init_random_prime(p, 512, 3);
    init_random_prime(q, 512, 7);
    mpz_init(n);
    mpz_mul(n, p, q);
    if (mpz_scan1(n, 1024 − 8) ≡ ULONG_MAX) {
      mpz_clear(n);
      mpz_clear(p);
      mpz_clear(q);
    }
    else {
      break;
    }
  }
```

This code is used in section 6.

8. Precomputing values for the Chinese remainder theorem

In order to speed up the signing function somewhat we precompute u_0 and v_0 such that $u_0p + v_0q \equiv 1 \pmod{n}$. We then store $u = u_0p$.

```
⟨ Chinese remainder precomputation 8 ⟩ ≡
  mpz_t u, v;
  xgcd(u, v, p, q);
  mpz_mul(u, u, p);
```

This code is used in section 6.

9. Generating a secret HMAC key

In the signing process we'll need to repeatably generate a random value. Thus we generate a random, 8 byte HMAC key here and store it as part of the secret key.

⟨ Generate HMAC secret 9 ⟩ ≡

uint8_t *hmac_secret*[8];

randombytes(*hmac_secret*, **sizeof** (*hmac_secret*));

This code is used in section 6.

10. Serialising a key pair

We serialise the keypair by writing p and q out as a series of little-endian 64-bit words. These values are, at most, 2^{512}, thus 8 such words is sufficient to describe them. u is, at most, 2^{1024}, so 16 words are sufficient for it. The value u may be negative so we use another byte of the private key to store its sign. Finally, we append the HMAC secret.

The public key is simply n and so 16 words are sufficient to describe it.

⟨ Keypair serialisation 10 ⟩ ≡

```
memset(sk, 0, SECRETKEYBYTES);
mpz_export(sk, Λ, −1, 8, −1, 0, p);
mpz_export(sk + 64, Λ, −1, 8, −1, 0, q);
mpz_export(sk + 128, Λ, −1, 8, −1, 0, u);
sk[256] = mpz_sgn(u) < 0 ? 1 : 0;
memcpy(sk + 257, hmac_secret, sizeof (hmac_secret));
memset(pk, 0, PUBLICKEYBYTES);
mpz_export(pk, Λ, −1, 8, −1, 0, n);
```

This code is used in section 6.

11. ⟨ Keypair cleanup 11 ⟩ ≡

```
mpz_clear(p);
mpz_clear(q);
mpz_clear(n);
mpz_clear(u);
mpz_clear(v);
```

This code is used in section 6.

12. The Extended Euclid function

This function calculates u and v from p and q such that $up + vq = \gcd(p, q)$. In this code, p and q are primes, thus $\gcd(p, q) = 1$. This is a very standard algorithm, see any number theory textbook for details.

On entry u and v should not have been initialised.

⟨ Extended Euclid 12 ⟩ ≡

```
static void xgcd(mpz_t u, mpz_t v, mpz_t ip, mpz_t iq)
{
  mpz_t p, q;
  mpz_init_set(p, ip);
  mpz_init_set(q, iq);
  mpz_init_set_ui(u, 1);
  mpz_init_set_ui(v, 0);
  mpz_t x, y;
  mpz_init_set_ui(x, 0);
  mpz_init_set_ui(y, 1);
  mpz_t s, t;
  mpz_init(s);
  mpz_init(t);
  while (mpz_sgn(q)) {
    mpz_set(t, q);
    mpz_fdiv_qr(s, q, p, q);
    mpz_set(p, t);
    mpz_set(t, x);
    mpz_mul(x, s, x);
    mpz_sub(x, u, x);
    mpz_set(u, t);
    mpz_set(t, y);
    mpz_mul(y, s, y);
    mpz_sub(y, v, y);
    mpz_set(v, t);
  }
  mpz_clear(p);
  mpz_clear(q);
  mpz_clear(x);
  mpz_clear(y);
  mpz_clear(s);
  mpz_clear(t);
}
```

This code is used in section 4.

13. Signing.

⟨ Signature generation 13 ⟩ ≡
  ⟨ Quadratic residue test function 26 ⟩
  ⟨ Signature compression function 28 ⟩
  ⟨ Signing function 14 ⟩

This code is used in section 1.

14. The signing function

This function takes a message in `m[0]`, `m[1]`, ..., `m[mlen - 1]` and a secret key in `sk[0]`, `sk[1]`, ..., `sk[SECRETKEYBYTES - 1]` and outputs a signed message in `m[0]`, `m[1]`, ..., `m[mlen + BYTES - 1]`.

⟨ Signing function 14 ⟩ ≡
```
int crypto_sign_rwb0fuz1024_gmp(uint8_t *sm, unsigned long long *smlen, const uint8_t
        *m, unsigned long long mlen, const uint8_t *sk)
{
  mpz_t p, q, u, v, n;
  ⟨ Import secret key 15 ⟩
  ⟨ Hash message 17 ⟩
  ⟨ Testing for residues 18 ⟩
  ⟨ Calculate tweaks 19 ⟩
  ⟨ Apply tweaks 20 ⟩
  ⟨ Pick root 21 ⟩
  ⟨ Calculate root 22 ⟩
  ⟨ Compress signature 23 ⟩
  ⟨ Export signed message 24 ⟩
  ⟨ Signing cleanup 25 ⟩
  return 0;
}
```

This code is used in section 13.

15. Importing the secret key

The secret key is serialised in the format that we used when generating the keypair. We import it and calculate $n = pq$ and $v = 1 - u$. (v and u were calculated such that $u + v \equiv 1 \pmod{n}$, u is a multiple of p and v is a multiple of q).

⟨ Import secret key 15 ⟩ ≡
```
mpz_init(p);
mpz_init(q);
mpz_init(u);
mpz_init(v);
mpz_import(p, 8, −1, 8, −1, 0, sk);
mpz_import(q, 8, −1, 8, −1, 0, sk + 64);
mpz_import(u, 16, −1, 8, −1, 0, sk + 128);
if (sk[256]) mpz_neg(u, u);
mpz_init(n);
mpz_mul(n, p, q);
mpz_set_ui(v, 1);
mpz_sub(v, v, u);
```

This code is used in section 14.

16. Hashing the input message

We need to turn the input message into an element in $\mathbb{Z}/pq\mathbb{Z}$.

Let $H_x(m)$ be a hash function from arbitrary length bytestrings to bytestrings of length x bits. Here $H_x(m)$ is defined as

$h_0 = \texttt{SHA512}(m \mid {}^{\#}\texttt{00000000})$

$h_1 = \texttt{SHA512}(h_0 \mid {}^{\#}\texttt{00000001})$

$h_i = \texttt{SHA512}(h_{i-1} \mid \mathit{u32be}(i))$

The h_is are concatenated until $>= x$ bits have been generated, then truncated to x bits. For example, for $H_{1024}(m)$, `SHA512` is run twice. This is very similar to `MGF1` from PKCS#1.

Convert the resulting bytestring into an element of $\mathbb{Z}/pq\mathbb{Z}$ by clearing the first byte and interpreting it as a big-endian number. Since we defined $pq > 2^{s-8}$, the result must be less than n.

There's a tiny chance that the result isn't in $\mathbb{Z}/pq\mathbb{Z}$, but this happens with probability $\approx 2^{-511}$ and we ignore it.

Call the resulting element $H(m)$.

This function calculates $H_{1024}(m)$ where m is in `m[0]`, `m[1]`, …, `m[mlen-1]` and returns the result in e, which should not be initialised on entry.

⟨ Hash function 16 ⟩ ≡

```
static void hash(mpz_t e, const uint8_t *m, unsigned mlen)
{
  uint8_t element[128];
  uint8_t counter[4] = {0};
  SHA512_CTX shactx;
  SHA512_Init(&shactx);
  SHA512_Update(&shactx, m, mlen);
  SHA512_Update(&shactx, counter, sizeof (counter));
  SHA512_Final(element, &shactx);
  counter[3] = 1;
  SHA512_Init(&shactx);
  SHA512_Update(&shactx, element, 64);
  SHA512_Update(&shactx, counter, sizeof (counter));
  SHA512_Final(element + 64, &shactx);
  element[0] = 0;
  mpz_init(e);
  mpz_import(e, 128, 1, 1, 1, 0, element);
}
```

This code is used in section 4.

17. ⟨ Hash message 17 ⟩ ≡

```
mpz_t elem;
hash(elem, m, mlen);
```

This code is used in section 14.

18. Testing $H(m)$ for residues

There's a 1/4 chance that $H(m)$ is a square in $\mathbb{Z}/pq\mathbb{Z}$. For it to be a square, it must be a square in both $\mathbb{Z}/p\mathbb{Z}$ and $\mathbb{Z}/q\mathbb{Z}$. This is easily tested since both prime orders are $\in 3+4\mathbb{Z}$, thus the root can be found by raising to $(p+1)/4$ and testing if the root squares to the correct result.

⟨ Testing for residues 18 ⟩ ≡

```
mpz_t pp1over4, qp1over4;
mpz_init_set(pp1over4, p);
mpz_add_ui(pp1over4, pp1over4, 1);
mpz_cdiv_q_2exp(pp1over4, pp1over4, 2);
mpz_init_set(qp1over4, q);
mpz_add_ui(qp1over4, qp1over4, 1);
mpz_cdiv_q_2exp(qp1over4, qp1over4, 2);
int a = is_quadratic_residue(elem, p, pp1over4);
int b = is_quadratic_residue(elem, q, qp1over4);
```

This code is used in section 14.

19. Calculating the tweak factors

We use two tweak factors, e and f, to make $H(m)$ a square where $e \in [1, -1]$ and $f \in [1, 2]$. By choosing e and f correctly, $efH(m)$ is a square. This is due to Williams ("A modification of the RSA public key encryption procedure", H. C. Williams, IEEE Transactions on Information Theory, Vol 26, no 6, 1980).

There are four cases: $H(m)$ may or may not be a square in each of $\mathbb{Z}/p\mathbb{Z}$ and $\mathbb{Z}/q\mathbb{Z}$. We write [Y,Y], for example, if $H(m)$ is a square in each.

$$(e, f) = \begin{cases} (1, 1) & \text{if [Y,Y]} \\ (-1, 1) & \text{if [N,N]} \\ (1, 2) & \text{if [N,Y]} \\ (-1, 2) & \text{if [Y,N]} \end{cases}$$

To see why, consider that in a group of prime order, if c is not a square, $-c$ is. Thus, if $H(m)$ is not a square in either prime group, $-H(m)$ is.

Also, 2 is a square in a group of prime order iff the order $\in 1+8\mathbb{Z}$ or $7+8\mathbb{Z}$. Since p is not such a prime, 2 is not a square, and non-square $*$ non-square is a square. Likewise, 2 is a square in $\mathbb{Z}/q\mathbb{Z}$ and square $*$ square is a square. Thus multiplying by 2 converts [N,Y] into [Y,Y]. For the same reasons it also converts [Y,N] into [N,N] since non-square $*$ square $=$ non-square.

⟨ Calculate tweaks 19 ⟩ ≡

```
int mul_2 = 0, negate = 0;
if (a ⊕ b) {
  mul_2 = 1;
  a ⊕= 1;
}
if (¬a) negate = 1;
```

This code is used in section 14.

20. Applying the tweaks

Once we have calculated e and f, we calculate $efH(m)$ and reuse the variable *elem* to store it.

⟨ Apply tweaks 20 ⟩ ≡

```
if (negate) mpz_neg(elem, elem);
if (mul_2) mpz_mul_2exp(elem, elem, 1);
if (negate ∨ mul_2) mpz_mod(elem, elem, n);
```

This code is used in sections 14 and 29.

21. Picking the root

Now that we have $efH(m)$, a square, we need to pick one of the four possible square roots modulo n. We need to pick the root in a random fashion, but it's vitally important that we pick the same root every time. If we were to generate different roots when signing the same message we leave ourselves open to attack.

Thus we calculate `HMAC-SHA512` of m using a secret value as the key and use the first byte of the result. Since the secret value is only known to us, no one else can calculate which root we pick and, since the secret value doesn't change, we'll always pick the same root for the same message.

The secret key was calculated when generating the keypair.

⟨ Pick root 21 ⟩ ≡

```
const uint8_t r = HMAC_SHA512(sk + 257, m, mlen);
```

This code is used in section 14.

22. Calculating the root

The most obvious method of finding a root of $efH(m)$ is to find a root in each of p and q (which we can do by raising to $(p+1)/4$) and combining them using the Chinese Remainder Theorem.

However, we wish to choose one of the four roots at random, so we use the bottom two bits of r to randomly negate the root in each of p and q before combining. Note that we precomputed values for the CRT calculation when generating the keypair.

Once we have done this we have a fixed, unstructured, $B = 0$ Rabin-Williams scheme and can use Bernstein's proof to show that a hash-generic attack against this scheme is equivalent to factoring. ("Proving tight security for Rabin-Williams signatures." Pages 70–87 in *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008, Proceedings*, edited by Nigel Smart, Lecture Notes in Computer Science 4965, Springer, 2008. ISBN `978-3-540-78966-6`.)

⟨ Calculate root 22 ⟩ ≡

```
mpz_t proot, qroot;
mpz_init_set(proot, elem);
mpz_powm(proot, elem, pp1over4, p);
mpz_init_set(qroot, elem);
mpz_powm(qroot, elem, qp1over4, q);
if (r & 1) mpz_neg(proot, proot);
if (r & 2) mpz_neg(qroot, qroot);
mpz_mul(proot, proot, v);
mpz_mul(qroot, qroot, u);
mpz_add(proot, proot, qroot);
mpz_mod(proot, proot, n);
```

This code is used in section 14.

23. Compressing the signature

Now we perform signature compression which is described later.

⟨ Compress signature 23 ⟩ ≡

```
mpz_t zsig;
signature_compress(zsig, proot, n);
```

This code is used in section 14.

24. Exporting the signed message

The signed message consists of 64 bytes of compressed signature, followed by the tweak bits, followed by the original message.

The tweak bits are encoded into a single byte where the LSB if 1 iff $e = -1$ and the next most significant bit is 1 iff $f = 2$.

⟨ Export signed message 24 ⟩ ≡
$memset(sm, 0, \texttt{BYTES} - 1);$
$sm[\texttt{BYTES} - 1] = (mul\_2 \ll 1) \mid negate;$
$mpz\_export(sm, \Lambda, -1, 1, 1, 0, zsig);$
$memcpy(sm + \texttt{BYTES}, m, mlen);$
$*smlen = mlen + \texttt{BYTES};$

This code is used in section 14.

25. ⟨ Signing cleanup 25 ⟩ ≡
$mpz\_clear(zsig);$
$mpz\_clear(n);$
$mpz\_clear(proot);$
$mpz\_clear(qroot);$
$mpz\_clear(pp1over4);$
$mpz\_clear(qp1over4);$
$mpz\_clear(elem);$
$mpz\_clear(u);$
$mpz\_clear(v);$
$mpz\_clear(p);$
$mpz\_clear(q);$

This code is used in section 14.

26. Testing for quadratic residues

A quadratic residue (often also called a 'square' in this document) is a number e such that there exists x where $x^2 \equiv a \pmod{p}$.

Since both our primes are $\in 3 + 4\mathbb{Z}$, we can test simply for this by calculating the square root $x = a^{(p+1)/4} \pmod{p}$ and then squaring it to check that $x^2 \equiv e \pmod{p}$.

This function returns non-zero iff e is a quadratic residue modulo p. *power* is equal to $\frac{p+1}{4}$.

⟨ Quadratic residue test function 26 ⟩ ≡

```
static int is_quadratic_residue(mpz_t e, mpz_t p, mpz_t power)
{
  mpz_t r, reduced_e;

  mpz_init(r);
  mpz_init(reduced_e);
  mpz_mod(reduced_e, e, p);
  mpz_powm(r, e, power, p);
  mpz_mul(r, r, r);
  mpz_mod(r, r, p);

  const int result = 0 ≡ mpz_cmp(r, reduced_e);

  mpz_clear(r);
  mpz_clear(reduced_e);
  return result;
}
```

This code is used in section 13.

27. HMAC function

HMAC is a standard cryptographic private-key signing function that we use as a random number generator when picking the signature root.

This function takes a key in `key[0]`, `key[1]`, ..., `key[7]` and a message in `value[0]`, `value[1]`, ..., `value[valuelen - 1]` and returns a single byte.

⟨ HMAC function 27 ⟩ ≡

```
static uint8_t HMAC_SHA512(const uint8_t *key, const uint8_t *value, unsigned valuelen)
{
  unsigned i;
  uint8_t keycopy[8];
  for (i = 0; i < 8; ++i) keycopy[i] = key[i] ^ #5c;
  SHA512_CTX shactx;
  SHA512_Init(&shactx);
  SHA512_Update(&shactx, keycopy, 8);
  SHA512_Update(&shactx, value, valuelen);
  uint8_t t[64];
  SHA512_Final(t, &shactx);
  for (i = 0; i < 8; ++i) keycopy[i] ^= #6a;
  SHA512_Init(&shactx);
  SHA512_Update(&shactx, keycopy, 8);
  SHA512_Update(&shactx, t, sizeof (t));
  SHA512_Final(t, &shactx);
  return t[0];
}
```

This code is used in section 4.

28. Compressing signatures.
A Rabin signature can be compressed to half its original size using continued fractions. This is due to Bleichenbacher ("Compressing Rabin Signatures", Daniel Bleichenbacher, Topics in Cryptology CT-RSA 2004, 2004, Springer, `978-3-540-20996-6`).

Bleichenbacher compression boils down to finding the demoninator of the principal convergent of s/n such that the demoninator of the next principal convergent is $> \sqrt{n}$.

The demoninators can be calculated with a recurrence relation: $v_{i+2} = v_{i+1} * c + v_i$ where c is the next element of the continued fraction expansion of s/n. Although we only need to keep track of three values for that recurrence relation, the code actually keeps track of four becuase $x \,\&\, 3$ is nicer than $x \,\%\, 3$.

This function takes a Rabin signature, s, the public value n and returns a compressed signature in *zsig*, which should not have been initialised upon entry.

⟨ Signature compression function 28 ⟩ ≡

```
static void signature_compress(mpz_t zsig, mpz_t s, mpz_t n)
{
  mpz_t vs[4];
  mpz_init_set_ui(vs[0], 0);
  mpz_init_set_ui(vs[1], 1);
  mpz_init(vs[2]);
  mpz_init(vs[3]);
  mpz_t root;
  mpz_init(root);
  mpz_sqrt(root, n);
  mpz_t cf;
  mpz_init(cf);
  unsigned i = 1;
  do {
    i = (i + 1) & 3;
    if (i & 1) {
      mpz_fdiv_qr(cf, s, s, n);
    }
    else {
      mpz_fdiv_qr(cf, n, n, s);
    }
    mpz_mul(vs[i], vs[(i - 1) & 3], cf);
    mpz_add(vs[i], vs[i], vs[(i - 2) & 3]);
  } while (mpz_cmp(vs[i], root) < 0);
  mpz_init(zsig);
  mpz_set(zsig, vs[(i - 1) & 3]);
  mpz_clear(root);
  mpz_clear(cf);
  mpz_clear(vs[0]);
  mpz_clear(vs[1]);
  mpz_clear(vs[2]);
  mpz_clear(vs[3]);
}
```

This code is used in section 13.

29. Signature verification.

This function takes a message signed in `sm[0]`, `sm[1]`, ..., `sm[smlen-1]` and verifies that it was signed by the public key in `pk[0]`, `pk[1]`, ..., `pk[PUBLICKEYBYTES-1]`. If the verification fails, it returns -1. Otherwise, the original message is written to `m[0]`, `m[1]`, ..., *mlem* is set to the length of the original message and 0 is returned.

```
⟨ Signature Verification 29 ⟩ ≡
  int crypto_sign_rwb0fuz1024_gmp_open(unsigned char *m, unsigned long long *mlen, const
          unsigned char *sm, unsigned long long smlen, const unsigned char *pk)
  {
    int res = 0;
    ⟨ Import values for verification 30 ⟩
    ⟨ Hash signed message 31 ⟩
    ⟨ Apply tweaks 20 ⟩
    ⟨ Verify compressed signature 32 ⟩
    *mlen = smlen − BYTES;
    memcpy(m, sm, *mlen);
  out: mpz_clear(zsig);
    mpz_clear(elem);
    mpz_clear(n);
    return res;
  }
```

This code is used in section 1.

30. ⟨ Import values for verification 30 ⟩ ≡

```
  if (smlen < BYTES) return −1;
  mpz_t n, zsig;
  mpz_init(n);
  mpz_import(n, 16, −1, 8, −1, 0, pk);
  mpz_init(zsig);
  mpz_import(zsig, 64, −1, 1, 1, 0, sm);
  const uint8_t negate = sm[BYTES − 1] & 1;
  const uint8_t mul_2 = sm[BYTES − 1] & 2;
```

This code is used in section 29.

31. ⟨ Hash signed message 31 ⟩ ≡

```
  mpz_t elem;
  hash(elem, sm + BYTES, smlen − BYTES);
```

This code is used in section 29.

32. Verifying a compressed signature

Now that we have calculated $efH(m)$, let v be the compressed signature, then let $t \equiv efH(m)v^2 \pmod n$. The signature is valid iff t is a square in $\mathbb{Z}$. An attacker can forge the signature for a message where $efH(m)$ is a square in $\mathbb{Z}$ but squares are around $2^{s/2}$ apart so this is infeasible unless the hash function is broken.

We also need to make sure that $\gcd(v, n) \neq 1$, otherwise an attacker could cause t to be 0 which is certainly a square. An attacker could choose $v \equiv 0 \pmod n$ or they could choose v to be a multiple of p or q. However, if they know p or q they have broken the system so that case is uninteresting. Thus, we actually need only check that $t \neq 0$.

⟨ Verify compressed signature 32 ⟩ ≡

```
mpz_mul(zsig, zsig, zsig);
mpz_mul(zsig, zsig, elem);
mpz_mod(zsig, zsig, n);
if (0 ≡ mpz_sgn(zsig)) {
    res = −1;
    goto out;
}
if (¬mpz_perfect_square_p(zsig)) {
    res = −1;
    goto out;
}
```

This code is used in section 29.

33. Acknowledgements.
Thanks to Daniel Bleichenbacher and Moti Yung for reviews and comments.

RWB0FUZ1024